Small-scale XMI programming: a revolution in UML tool use?

Perdita Stevens
Division of Informatics
University of Edinburgh

1 Introduction

UML, the Unified Modeling Language, is a standard diagrammatic language for recording the design of systems, especially object-oriented software systems. One of the main benefits of a *unified* modeling language is that it enables competition between tool vendors and allows users a wide choice of tools. Getting the most out of a tool - which is often a significant investment - means using it as more than a fancy drawing tool. In this position paper I will argue that the combination of XML and UML - especially in the form of XMI, the OMG's standard XML Metadata Interchange format - is crucial enabling technology for getting good value out of a UML tool. Indeed, I believe that it may *revolutionise the use of modelling tools in future.*

2 Small-scale tool development and integration

Let us compare the facilities long available to the programmer with those available to the designer until now. Coders have the culture that they are in control; their tools are there to assist. The artefact they are producing - the code - is visible and available to them as plain text, so all text-manipulation facilities can be applied to code. For example, the developer can load the code into a more or less intelligent editor and hack it about, either with a specific plan or in an attempt to understand it; they can apply editing macros; they can write scripts in their favourite text manipulation language, e.g. Perl, to do what they will with the code. Any professional software engineer is used to doing this kind of manipulation on their code, and sometimes on other people's, on an ad hoc basis. For example, some of the tasks I've carried out personally are:

- "grep"ing for particular relevant identifiers, or for other patterns, in order to find relevant sections of code quickly;
- writing small scripts to check for, and in some cases correct, errors not caught by the type checker of the language, or for violations of coding standards;
- extracting comments and formatting them for use in an in-house, non-standard help system whose use was mandated;
- extracting information about method signatures etc. for insertion in (LaTeX, in the case in point) documentation.

Any of these things could be done by hand, but, given the appropriate tools, can be automated *so easily that the effort of automation is often justified even for a one-off instance of the task.* Thus they fall into the category of mini-tools that may be developed by any developer, not just by those designated as project toolsmiths, let alone by specialised tool companies.

Note that an interesting feature of ad hoc utilities written for immediate use by the developer who wants them is that it is possible for them to be much simpler than a generic tool written by a third party to do the same task could be, because they can take advantage of any simplifying assumptions that can be made about *this particular* problem, e.g. things the developer happens to know about the nature of the code to which the script is to be applied. We will take full advantage of this in our examples (which are in any case simple for presentational reasons). As always, it is important that everyone involved understands the difference between an ad hoc script that solves the problem at hand and a robust tool that will work or fail gracefully in a wide variety of circumstances.

When UML first appeared, there was no standard format for interchange of UML models; most individual tools had their own textual format which you could reverse engineer if you really wanted to hack it, but the results tended to be unpredictable, because the formats had not been designed for such use. XMI, the OMG's XML Metadata Interchange format, is a vendor-independent format for saving and loading UML models. (Some incompatibilities between XMI written by different tools still exist, but this will settle as XMI and the tools mature. All XMI files used with the scripts described here are written by Argo/UML 0.8 (`http://www.argouml.org`.) So far most of the interest in XMI has come from tool vendors and has inevitably been for rather heavyweight tool integration (see e.g.[1]), though a partial exception is [2]. In this

paper we argue that, important as this is, *the main impact of XMI may be for lightweight tool development and integration of the kind that can be done in minutes or hours, rather than days, weeks or months, by any developer.*

XMI is not an easy format for a human to read, and even small models can translate into large XMI files. However, the big advantage of XMI being based on XML is that the whole range of generic XML tools is available. Developers writing scripts to work on code generally avoid the need to parse the code, but scripts working on XMI can easily take advantage of parse tree information, because XML parsers are available in every popular language. The ability to analyse and manipulate XMI files means:

- Analyses or changes that used to be tedious to do by hand using the GUI of a UML tool can be automated; and so the temptation to let the UML model get out of step with the code is decreased. For example, changes made to the model may propagate to the code using tools' own forward/reverse engineering combinations, so that a developer may choose to make a change to the model and propagate it to the code rather than just changing the code.
- Any developer can write a script to extract information from XMI files and turn it into the input format of a proprietary tool they may be familiar with.

We will illustrate each.

3 Model analysis example

Any technology should make simple things simple. To start at the simplest conceivable level, consider a case in which a developer wishes to change some identifier wherever it occurs in the model. This can be done by loading the XMI file into any reasonable editor and applying search-and-replace capabilities, with or without querying the user for confirmation; or alternatively, by a script in any text manipulation language, such as Perl. The reason why this is worth saying at all is to illustrate that the problem faced by a developer wanting to manipulate a particular model is simpler than that faced by a tool vendor wanting to allow such editing capabilities within their tool. For example, the developer may be confident that a simple "replace retrieve by get" will do, whereas a vendor writing a capability that should work on any model would have to worry more about the other contexts in which "retrieve" might occur and whether the replaced "get" version will clash with anything already in existence. The addition of human intelligence from the developer makes the problem soluble with much less programmed intelligence.

To take a slightly less trivial example, demonstrating what can be done with an XML parser, consider the case in which a developer wants to find all public attributes (but not operations) of classes, perhaps prior to replacing them with non-public attributes with get/set methods. If the model is large this can be a tedious task to do graphically with a UML tool, or even with an unfamiliar proprietary scripting language. Figure 1 shows a Perl script that accomplishes it.

However daunting this may look to non-Perl-devotees, this is quite straightforward to someone who knows Perl; it's the kind of thing that could be written in minutes (especially if this is not the first time the developer has used the XML::Parser module) and might save effort even if it was used only once. As prefigured above, this script takes advantage of hypothesised simplicities of the target model, e.g., it takes no account of packages. Note also that the only component used is a simple XML parser. If one were going to do a lot of this kind of thing it would probably be worth writing a specialist XMI package, but even without one the script is only 42 lines long. It would not, of course, be hard to extend the script in various ways, for example to modify the XMI rather than merely examining it, if this was required.

4 Tool integration example

In this section we aim to back up our claims by giving an example of how a UML tool that exports XMI can simply be integrated with another tool. The aim of the example is not to do clever tool integration - reports of much more impressive integrations have been published - but to show that something that may be worthwhile in context can be done very easily.

A script very similar to the one shown above (but a little longer: 136 lines; see `http://www.dcs.ed.ac.uk/home/pxs/XSE2001/state.pl`) is used to extract the structure of any single flat FSM-like state diagram (no concurrency, actions etc.; though again, extending the script to more features would not be hard) and translate it into a Calculus of Communicating Systems (CCS) agent described in the input format of the Edinburgh Concurrency Workbench (CWB, see `http://www.dcs.ed.ac.uk/home/cwb`). The CWB is a broad verification tool (over 60 commands) in which many kinds of analysis can be carried out on such CCS agents, including model checking the full modal mu calculus with game-based feedback and many versions of equivalence and preorder checking.

Finally, Figures 2-4 show an example of a UML state diagram drawn in Argo/UML, the CWB input file produced by the script from the saved XMI file, and a few CWB commands carried out on that input file (comments in [square brackets]).

Figure 1: Identifying public attributes

```
#!/usr/local/bin/perl
use XML::Parser;
my $file = shift;
die "Can't find file \"$file\"" unless -f $file;
my $parser = new XML::Parser(Style => Tree, ErrorContext => 2);
my $pairref = $parser->parsefile($file);

$VISIBILITY = 'Foundation\.Core\.ModelElement\.visibility';
$CLASS = 'Foundation\.Core\.Class$';
$NAME = 'Foundation\.Core\.ModelElement\.name$';
$ATTRIBUTE = 'Foundation\.Core\.Attribute';

mypair(undef, undef, @$pairref);

# take hashrefs for current class and attribute, plus tag-content pair
sub mypair {
  my ($recclass, $recattr, $tag, $content) = @_;
  return $content unless $tag; # deal with non-element nodes, i.e. text
  if ($tag=~/$ATTRIBUTE/) {
    my $attr = {}; # the attr we've found, rep by new ref to anon hash
    myarray($recclass, $attr, @$content);
    print "Attr $$attr{name} in class $$recclass{name} is public\n"
      if $$attr{visibility} =~ 'public';
  }
  elsif ($tag=~/$VISIBILITY/ && $recattr)
    { $$recattr{visibility} = ${$$content[0]}{'xmi.value'}; }
  elsif ($tag=~/$NAME/ && $recattr){$$recattr{name} = @$content[2];}
  elsif ($tag=~/$CLASS/) {myarray({}, $recattr, @$content);}
  elsif ($tag=~/$NAME/) {
    $$recclass{name} = @$content[2] unless $$recclass{name};
  }
  else { myarray($recclass, $recattr, @$content); }
}

# take hashrefs for current class and attribute, plus an element content, viz
# a hashref for attributes, which we ignore, followed by (tag,content)*
sub myarray {
  my ($class, $attr, $attributes, @rest) = @_;
  while (@rest) {
    mypair ($class, $attr, shift @rest, shift @rest);
  }
}
```

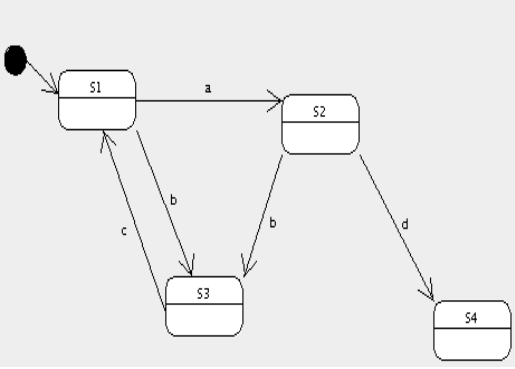
Figure 4: Using the CWB to analyse the behaviour

```
Edinburgh Concurrency Workbench, version 7.1,
Sun Jul 18 21:19:30 1999
Process algebra: CCS
Optional modules in this build: AgentExtra,Graph,Divergence,Contraction,
Equivalences,Logic,Simulation,Testing
Command: input "foo.cwb";
Command: deadlocks S1;
--- a d ---> S4
Command: deadlocks S3;
--- c a d ---> S4
Command: checkprop (S1, max (X.<->X));
true [i.e. there is an infinite path from S1]
Would you like to play (and lose!) a game against the CWB? (y or n) y
[output omitted]
Command: agent T = a.b.c.T + b.c.T;
Command: dftrace (S1,T);
Agent
S1
can perform action sequence
a,d
which agent
T
cannot.
Command:
```

Figure 2: CWB input file (foo.cwb) produced by script

```
agent S3 = c.S1;
agent S4 = 0;
agent S1 = a.S2 + b.S3;
agent S2 = b.S3 + d.S4;
```

Figure 3: State diagram for class Foo, drawn in Argo/UML



5 Conclusion

We have demonstrated that the use of XMI can make it easy to carry out small tasks on (possibly large) UML models. In particular, such techniques can make tool integration available to developers in a way which is quite new. Whilst we acknowledge the danger that such rapidly written ad hoc scripts may contain bugs, in the context of tasks carried out by the developer who writes the script the risk is likely to be acceptably small, especially when the alternative is to carry out the same task by hand: carrying out repetitive, boring tasks on large models is also error prone! Further work will focus on the development of lightweight utilities, patterns and methods for maximising the benefits and minimising the risks of these techniques.

Acknowledgements

I would like to thank the referees, and the EPSRC for financial support in the form of an Advanced Research Fellowship.

REFERENCES

[1] Steve Brodsky. Xml metadata interchange. `http://www-4.ibm.com/software/ad/standards/xmi.html/xmiwhite0399.pdf`.

[2] C. Nentwich, W. Emmerich, A. Finkelstein, and A. Zisman. Box: Browsing objects in xml. *Software Practice and Experience*, 30:1–16, 2000.